



Microservices & Event-Driven Architecture

⌚ Creada por	 Hernández Tomás
⌚ Hora de creación	@10 de agosto de 2025 2:12
☰ Categoría	
⌚ Última edición por	 Hernández Tomás
⌚ Fecha de última actualización	@13 de enero de 2026 0:48

Microservicios

Motivación a Microservicios

Es la arquitectura más moderna, actual y popular en la industria. En este tipo de arquitectura, todo nuestro sistema se organiza como una colección de servicios independientes.

Cada servicio tiene su propia responsabilidad. Este tipo de arquitectura es fundamental en muchas empresas grandes como Amazon, Google, Netflix, Meta, Uber, Airbnb, etc.

Cuando los microservicios se implementan bien, permiten a las organizaciones:

1. Escalar rápidamente.
2. Alcanzar millones de usuarios.
3. Mantener bajos los costos operacionales.
4. Permite ser eficientes e innovar.

No siempre es la mejor opción

La idea de usar microservicios y escalar rápido puede ser motivador.

Sin embargo, muchas organizaciones los implementan mal y deshacen la decisión de migrar pero si se aplican bien (y en el momento adecuado) **pueden ser muy beneficiosos.**

Qué vamos a ver acá

1. Precondiciones para usarlos y tener éxito.
2. Como migrar un monolito a microservicios.
3. Mejores prácticas (Google).
4. Industry-proven patterns.
5. Testing.
6. Deploying Microservices to production.
7. Troubleshooting (Observability)

Al final del curso seremos capaces de tener la intuición de saber qué debe ser un microservicio y qué no.

Además, tendremos conocimientos para evitar **errores, pitfalls, anti-patterns.**

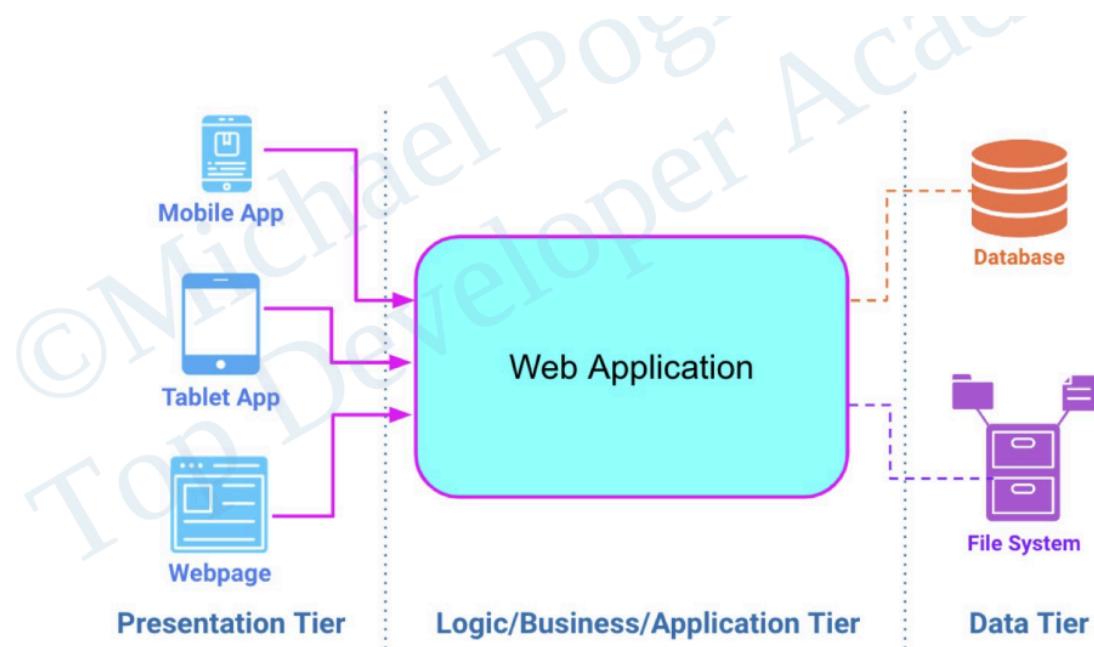
Saber **microservicios es esencial para un senior en empresas como Google en entrevistas de System Design**

Motivación a Event-Driven Architecture

Comúnmente se usa con la arquitectura de Microservicios estableciendo una comunicación asíncrona basada en eventos entre microservicios, lo cual nos permite conseguir un desacoplamiento aún mayor y una mayor escalabilidad para nuestra empresa.

Además, permite implementar patrones poderosos para los microservicios.

Monolithic Architecture



En una aplicación monolítica, la lógica de negocio, el desarrollo del backend y el manejo de la base de datos está en una misma aplicación (arquitectura de tres niveles).

Este tipo de arquitectura tiene muchos beneficios y es por eso que hoy en día se sigue utilizando.

Los **beneficios** que proporciona esta arquitectura son:

1. Fáciles de diseñar. Se acopla a cualquier sistema web, sea cual sea el tipo del negocio.
2. Fácil de implementar: Los desarrolladores no tienen que romperse la cabeza. Usan tecnologías que conocen y podemos tener fácilmente un sistema funcional.

La arquitectura monolítica se usa en **startups** y en **equipos pequeños**.

Hay un dicho que dice que si tu equipo puede ser alimentado por 2 pizzas, entonces podés seguir usando arquitectura monolítica.

Las **desventajas** de esta arquitectura son:

1. Poca escalabilidad organizacional: Al haber tantos desarrolladores en un mismo codebase, se hace común ver problemas de MRs, muchos conflictos. Se pisan unos con los otros.

2. Código complejo: A medida que agregamos código y código, se hace más grande y complejo. Esto produce que sea más difícil de razonar, el IDE tarda más en cargar, es más lento para buildar y testear. Más riesgos para deploy, hay features más largas y menos deploys. También contribuye a que el onboarding a nuevos desarrolladores sea complejo.
3. Poca escalabilidad del sistema: Cada instancia de aplicación que contiene toda la lógica requiere más CPU y memoria. Tenemos que correr cada instancia en computadoras más caras. También es más difícil de migrar algunas tecnologías viejas. Cualquier bug pequeño o de rendimiento afecta a toda la aplicación y nos obliga a hacer un redeploy entero (o rollbacks).
4. Estamos atados a usar un mismo lenguaje, tecnologías para todo el negocio: Esto no suele ser bueno porque hay algunos procesos que requieren de otros paradigmas o incluso lenguajes que performan mucho mejor para una tarea dada.

Formal: Microservicios

Organiza la lógica empresarial como una colección de servicios poco acoplados y desplegados individualmente.

Cada servicio pertenece a un pequeño equipo y tiene un ámbito de responsabilidad limitado.

Beneficios

1. Obtenemos una escalabilidad organizativa mucho mayor ya que cada servicio contiene un subconjunto de la funcionalidad global, por lo que el codebase de cada servicio es menor. Esto permite que el código cargue mas rápido en un IDE.
 - a. Fácil de testear cada uno y de entender qué hace.
 - b. Onboarding más rápido para engineers que tengan que mantenerlo.
 - c. Los equipos son más veloces trabajando en un “solo mundo”.
2. Escalabilidad del sistema más alta
 - a. En una arquitectura monolítica cada servidor tiene toda la aplicación y conlleva un costo más alto de mantener. En microservicios, cada instancia es mucho más chica, por lo que cada instancia de un

microservicio consume menos memoria y CPU, lo cual puede ser corrido por servidores más baratos.

- i. Si algún microservicio necesita más recursos, le podés asignar a su contenedor más potencia de CPU o más ram (escalado vertical).
- b. Cada microservicio puede estar implementado con un lenguaje diferente. Ellos pueden refactorizar (*o cambiar el lenguaje según necesiten*) de manera más eficaz.
- c. Mayor estabilidad para todo el sistema: Si algún microservicio falla podemos levantar más instancias de ese rápidamente sin tener que esperar un deploy entero de un codebase gigantesco.

Ojo: En Microservicios, normalmente metés en un mismo servidor varios contenedores (microservicios), y podés escalar cada uno verticalmente según necesites.

A un microservicio le podés asignar 2GB de ram, al otro 4GB, y así sucesivamente. En este contexto, el escalado vertical refiere pasar de 2GB de ram a 8GB en un contenedor.

No es la misma definición de cuando hablamos de cambiar literalmente el hardware (mejorar el procesador o agregar más ram).

Desafíos

1. Los microservicios son un **gran sistema distribuido**. En la arquitectura monolítica tenemos un comportamiento, éxito y rendimiento predecible pues cuando llamamos a algo que está dentro del mismo sistema es sencillo de trackear.
 - a. En los microservicios esto no sucede, el comportamiento, éxito y rendimiento no es tan predecible porque la comunicación es entre **computadoras diferentes**. Suele haber latencia, pérdida de paquetes o inclusive errores.
2. Testing: no hay garantía de que cuando todos los servicios estén en producción funcionen entre ellos (integration tests). Lo complicado es saber qué microservicio es el responsable de hacer esos tests de integración.
3. Dificultad de monitorear rendimiento y bugs: Si un cliente hizo una solicitud para hacer X cosa pero esa X cosa necesita comunicarse con 10

microservicios para responder es muy difícil de trackear qué sucede si algo falla.

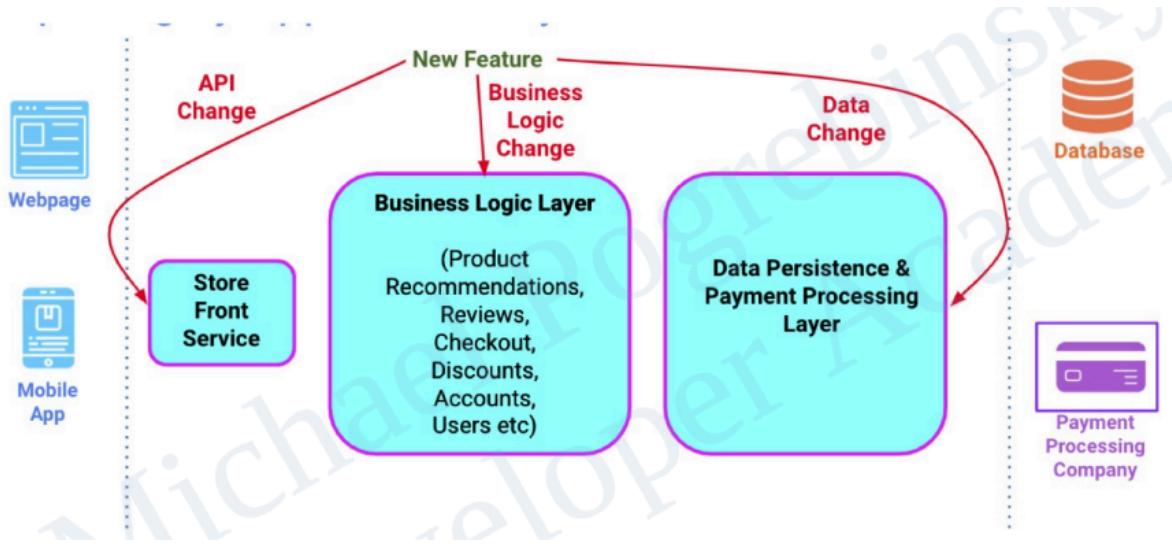
- a. Puede suceder que la información no llegue.
 - b. Puede suceder que la información sea incorrecta.
 - c. Puede suceder que la información demore demasiado tiempo.
4. Dificultad en la separación de responsabilidades: Si las responsabilidades de los microservicios están mal distribuidas podemos tener problemas organizacionales. Si un cambio en un microservicio conlleva a hablar con otros equipos hay algo raro (cada equipo debería funcionar por sí solo).

Si estos puntos no se tienen en cuenta, en vez de armar una arquitectura de microservicios habremos construido un “monolito distribuido”.

Migrar arquitectura monolítica a arquitectura microservicios

Intento 1 (separar por capas de la aplicación)

Separar un codebase en un set arbitrario de microservicios no nos va a dar ningún beneficio.



Estas tres capas de Store Front Service (middlewares, permisos) + Business Logic Layer (Checkout, Discount, Accounts, ...) + Data Persistence & Payment Processing Layer (Mongo, Postgres,...) eran una sola.

Separarlas de esta manera no tiene beneficios porque si agregamos una nueva funcionalidad, necesitamos modificar los 3 para acoplar este cambio.

Recordemos que la idea de los microservicios es que si queremos agregar algo, debería bastar con cambiar un único microservicio sin siquiera el resto enterarse.

Lo cual este primer intento de separar un monolito en microservicios es **incorrecto**. No basta con separar porque sí. Este intento nos produjo una arquitectura con 5 monolitos. Entonces ¿cuál es el límite de cada microservicio? Hay un principio que lo define.

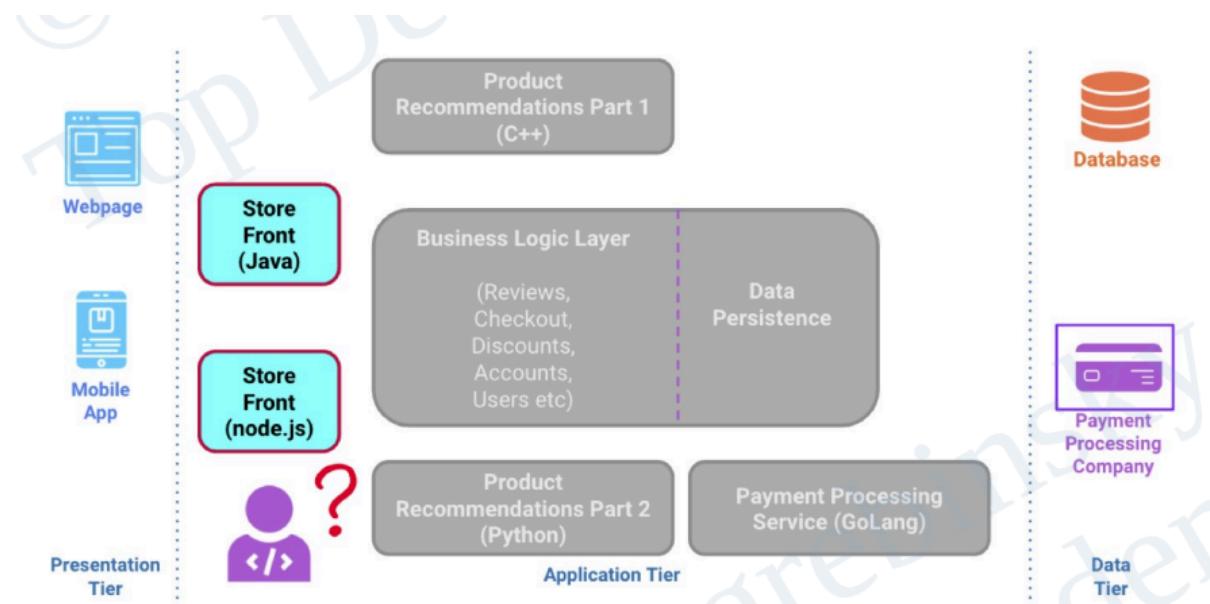
Microservices Boundaries (límites) - Core Principles pt 1

Cohesion: Los elementos que están muy relacionados uno con el otro y cambian constantemente cuando implementamos algo deben **seguir estando juntos**. Esto significa que esa pieza del sistema debería ser un único microservicio.

Nuestro intento anterior falló porque no cumplió la cohesión.

Intento 2 (separar por límites tecnológicos)

Imaginemos que tenemos cuellos de botella por algunos procesos y decidimos cambiar nuestro enfoque y reescribir cada proceso en un lenguaje diferente.



El problema de separar por **tecnología** no nos da beneficios.

¿Por qué? Porque si necesitamos implementar un feature no queda claro qué equipo es responsable de la implementación.

¿Para qué tenes un Product Recommendations en C++ y otro en Python? ¿En qué se difieren? ¿Cuál es el límite de cada uno? ¿Por qué?

No tiene sentido que dos cosas hagan lo mismo pero estén escritos en un lenguaje diferente.

Microservices Boundaries (límites) - Core Principles pt 2

Single Responsibility Principle (SRP): Cada componente debe hacer **solamente** una cosa y hacerla **bien**.

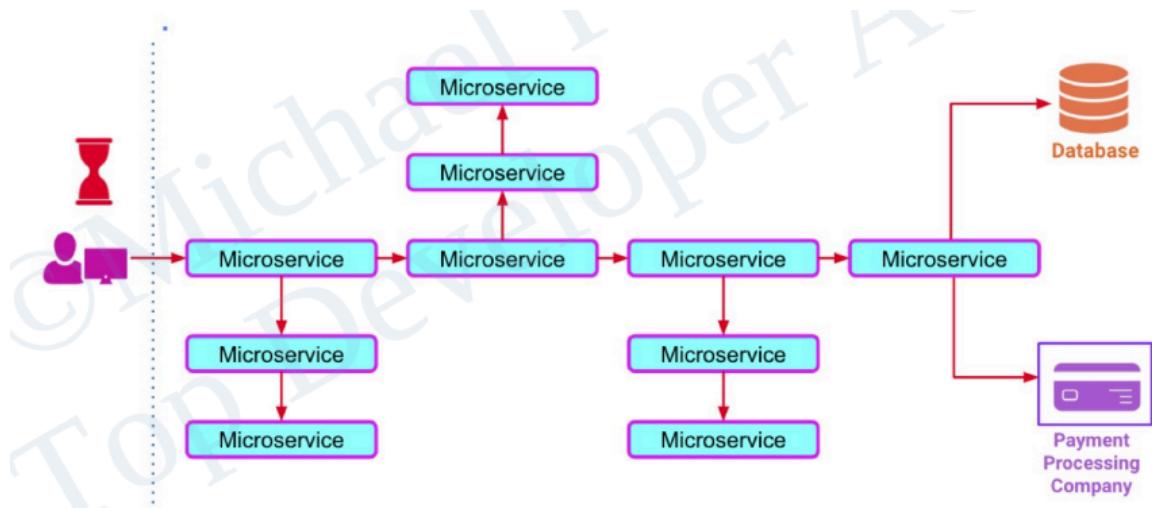
Cumplir este principio elimina toda ambigüedad con respecto a responsabilidades. Además, la terminología utilizada en el microservicio está atada a ese contexto.

Por ejemplo, si estás leyendo el microservicio de Users y tenés un getById sabés que es obtener un User por id.

Intento 3 (minimizando el tamaño de los microservicios)

A veces se asume que solamente por separar y disminuir el tamaño de los componentes todo va a ser mejor.

Acá no es cierto pues cuantos más microservicios tenemos todo va a ser más lento por la naturaleza de la comunicación entre computadoras.



¿Qué estamos incumpliendo acá? Que todos los microservicios en realidad están totalmente atados entre sí.

Si una petición para ser resuelta necesita de que se conecten **todos los microservicios** o la **mayoría** algo está mal.

Microservices Boundaries (límites) - Core Principles pt 3

Loose Coupling: como máximo debe de existir una pequeña dependencia entre los microservicios.

Cada operación debe de requerir pocas comunicaciones con el exterior.

Pre-requisitos para una arquitectura de Microservicios exitosa

El tamaño del microservicio no es importante.

Lo que importa es que los microservicios cumplan:

1. Cohesividad: juntar las responsabilidades relacionadas en un único microservicio.
2. **Single Responsibility Principle:** cada microservicio hace una sola cosa y la hace excelente
3. **Loosely coupled microservices:** se deben de requerir mínimas comunicaciones entre microservicios para resolver una petición dada.

Descomposición de una aplicación Monolítica a Microservicios

Descomposición por Business Capabilities

Analizamos nuestro sistema desde un punto de vista 100% del negocio.

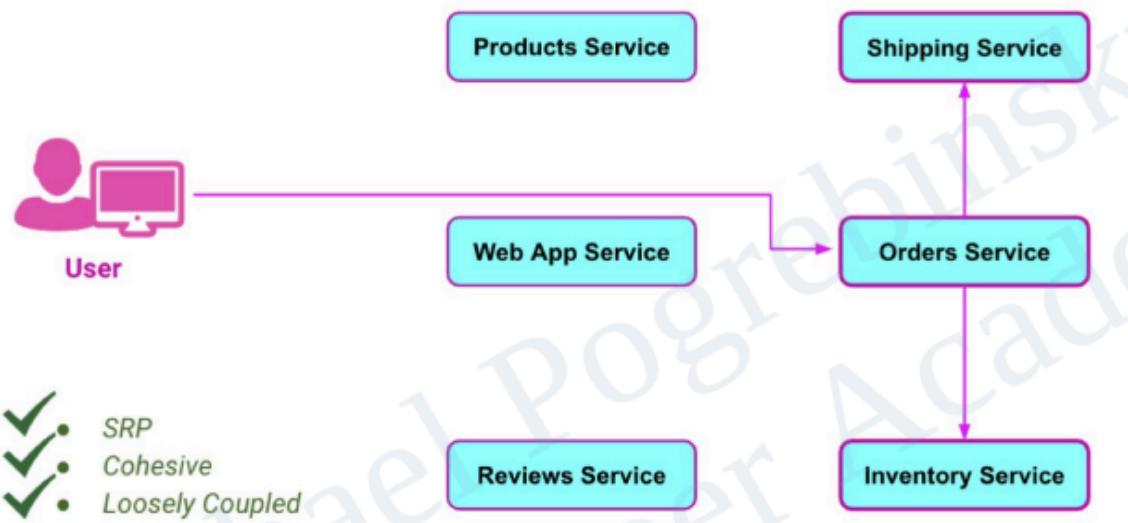
¿Cómo podemos identificar las capacidades?

1. Describiendo el sistema a una persona no-técnica.
2. Explicar qué hace el sistema y qué valor aporta esa capacidad.

Por ejemplo, en una tienda online tenemos las siguientes capacidades

1. Buscar mediante productos: Web App Service

2. Buscar/ver productos: Products Service
3. Leer reviews: Reviews Service
4. Hacer una orden: Orders Service
5. Entregar el producto: Shipping Service
6. Actualizar/Mantener el inventario de productos: Inventory Service



Una vez que separamos estas capacidades hay que preguntarnos si cumplen los 3 principios fundamentales.

1. SRP: Sí. Por contexto, cada cosa tiene su nombre y no necesita del otro.
2. Cohesive: Sí. Porque todas las responsabilidades que están relacionadas están en el mismo microservicio.
3. Loosely Coupled: Sí, porque cuando queremos comprar un producto pasamos por el OrdersService y no por Reviews por ejemplo. Si queremos hacer una Review vamos solo al Reviews Service.

Descomposición por Domain / Subdomain

El límite de cada microservicio lo determina el punto de vista de los ingenieros del equipo con el objetivo de que sea más intuitivo para ellos.

Los microservicios se separan en varios grupos en varias

1. Core: Es la parte central y diferenciadora del negocio, aquello que hace que la empresa sea única. Por ejemplo, en una tienda online, mostrar los

productos es indispensable para que alguien lo adquiera.

2. Supporting: Son partes que ayudan a entregar el valor del núcleo. Por ejemplo, generar órdenes o pagos para los clientes que quieren un producto determinado. El pago o la orden no existen sin producto.
3. Generic: Funcionalidades que no son específicas del negocio, que pueden comprarse o usarse como servicios externos. Por ejemplo, monitorear los microservicios con una librería.

¿Cuáles son las ventajas de este enfoque? Que sabés a donde priorizar los esfuerzos, ponés los mejores ingenieros en el core, ahorrás costos/tiempos sabiendo qué es prioridad y qué no.

Online Store - Example

Core Subdomains

- Products Catalog

Supporting Subdomains

- Orders
- Inventory
- Shipping

Generic Subdomains

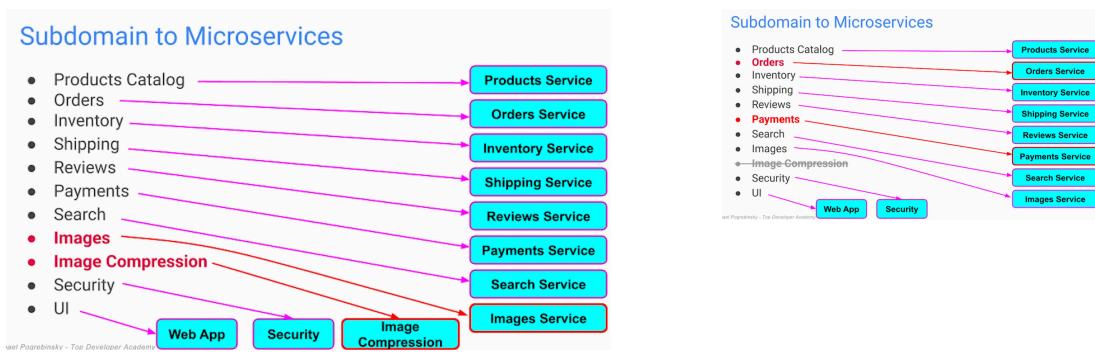
- Reviews
- Payments
- Search
- Images
- Image Compression
- Security
- Web UI

Sin productos no hay tienda.

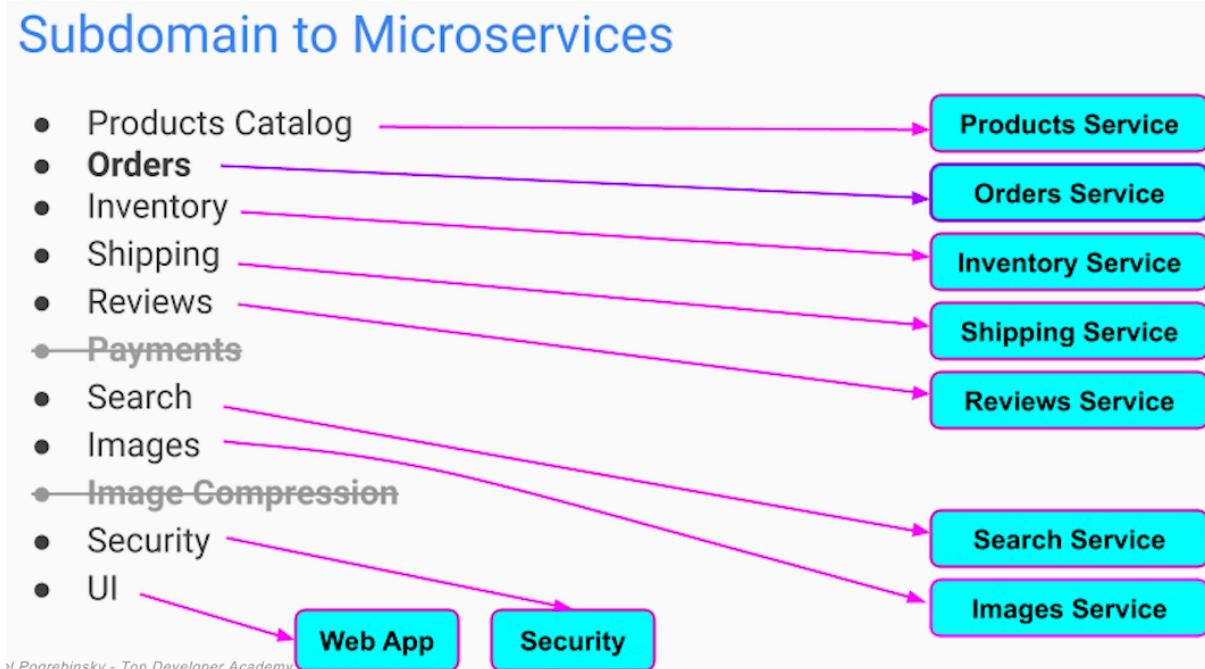
Sin órdenes, inventario o envío no hay venta.

Sin Reviews no hay confianza.

Notemos que Images e Image Compression están separados, pero hacer dos microservicios para ambos sería una mala idea porque están muy relacionados. Así que podríamos juntarlos.



Lo mismo con las órdenes y pagos. Si vemos que los pagos no aportan mucho a nivel lógico y requerimos mucho de las órdenes, sería mejor juntarlos en uno solo.



¿Qué forma elegir? ¿Por Business Capabilities o Subdomains?

	By Business Capabilities	By Subdomain
Cohesion and Loose Coupling	Winner	
Size of Microservices		Winner
Stability of the Design	Winner	
Intuitive for Engineers		Winner

La respuesta es: no hay una manera de decir usá esta porque esta no sirve. Ambas sirven, pero cada una tiene su ventaja y desventaja.

Migrar arquitectura a Microservicios (manos a la obra)

Big-Bang Approach

Plan:

1. El equipo delimita los límites de cada microservicio.
2. Se **detiene** el desarrollo de features hasta que la migración a microservicios finalice.

En la superficie:

1. Parece una buena idea (para mi si es muy grande no, porque si se te rompe algo o la gente pide más cosas te vas a retrasar mucho y quizás aparece otro competidor - exacto).

En la realidad:

1. Es el peor enfoque. Impacta mucho en el negocio.
 - a. Lo que queremos evitar en microservicios es que todos estén trabajando en lo mismo. Esto los junta y obliga.
 - b. Difícil de trackear el esfuerzo para un proyecto **tan largo y ambiguo**.
 - c. Si el manager está apurado produce que la migración a microservicios sea desestimada y el trabajo realizado hasta ese momento habrá sido en vano.
 - d. Parar el desarrollo de features hace que otros competidores aparezcan y te quiten clientes. Los Product Managers se aburrirían si ellos sacan

ideas y nadie las implementa.

Incremental and Continuous Approach

Plan: Identificar qué componentes serían los más beneficiados de estar en un microservicio.

¿Cómo identificamos cuáles son los componentes más beneficiados?

1. Las áreas que tienen más desarrollo frecuente y cambios.
2. Componentes que tengan requisitos de **alta escalabilidad y rendimiento** (los aislás para que hagan lo importante lo más rápido posible).
3. Componentes que tengan la menor deuda técnica (tener que migrar aquellos que no son muy claros o son muy difíciles de mantener, suelen ser un dolor de cabeza)

La mejor área para tomar en cuenta es la primera.

Esto se hace por cada posible separación del monolito.

Beneficios:

1. No hay que determinar deadlines para procesos gigantescos.
2. Es consistente, visible y permite que el progreso sea trackable.
3. El negocio no se abandona.
4. Si el proceso se atrasa no hay mucho problema, no estamos dejando nada de lado por esto.

Preparación para la migración

1. Crítico: Agregar tests en el código actual del servicio monolítico a migrar. Cuando terminemos la migración es nuestra obligación garantizar el mismo funcionamiento.
2. Definir una API para el microservicio.
3. Aislar este microservicio removiendo las interdependencias con el resto (para cumplir los tres puntos esenciales de arquitectura de microservicios).

Ejecutar la migración usando el Strangler Fig Pattern

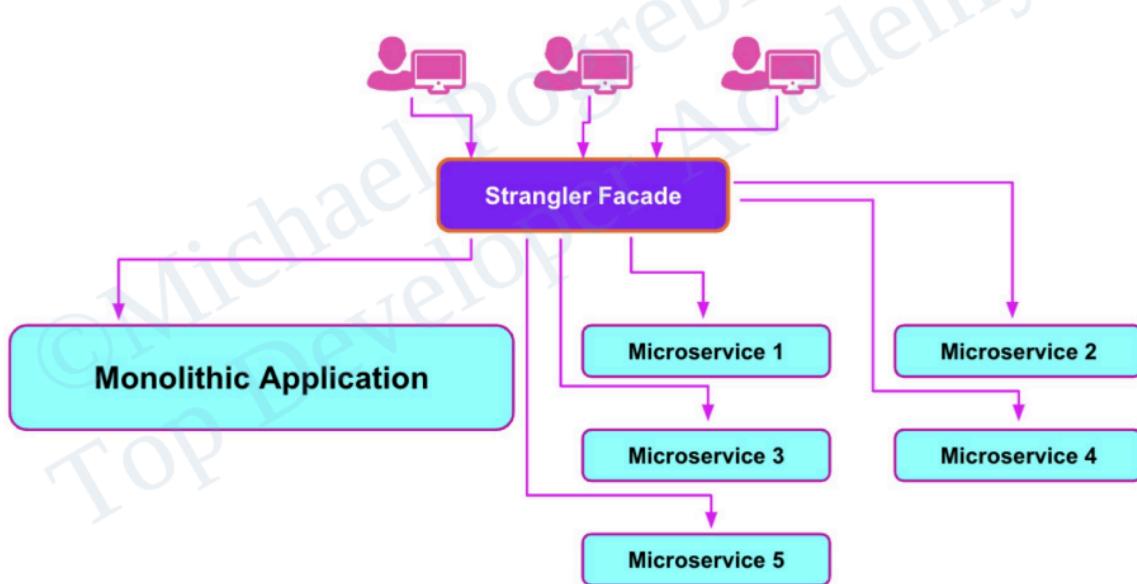
Es lo que hago yo siempre yo

La idea es tener cliente → API Gateway → aplicación monolítica.

Si el día de mañana ya tenés el microservicio funcional, entonces con el proxy (API-GATEWAY) redirigís ahí el tráfico que corresponda.

La gente no se va a enterar que ahora fue a un microservicio en vez de a un monolito.

Esto se hace cada vez que levantás un microservicio nuevo, y lo hacés hasta que terminás.



Migraciones smooth

La primera vez que pases un componente de monolito a microservicio, mantené el microservicio con el mismo lenguaje que estaba en el monolito.

Esto te va a ahorrar dolores de cabeza en temas de librerías / sintáxis / etc. Además te permite mitigar el riesgo de que no se produzcan cosas inesperadas.

Una vez que tu microservicio es estable y funciona, y ahora necesitás que sea más rápido (de lo que te permite el lenguaje), entonces agregá tests a tu microservicio y repetí el proceso de arriba.

Reescribí el microservicio en la tecnología que cumpla tus requerimientos, y hacé lo mismo que antes.

Servicio Monolítico → Microservicio → Microservicio en otro lenguaje (más rápido)

Ejercicios

You're a software architect at a company that migrated from Monolithic Architecture to Microservices Architecture.

You noticed that anytime a new feature is added to **Microservice A**, it requires a code change in **Microservice B**.

What is the most correct statement?

This is an indication that Microservice A is tightly coupled to Microservice B.

This is an indication that Microservice A is not cohesive enough.

Correcto

That's right! Reminder: Cohesion means code that changes together stays together.

This is an indication that our company was not ready for the migration to Microservices Architecture and we should migrate back to a Monolithic Architecture.

Si cada vez que cambiás algo en A necesitás cambiarlo en B significa que están súper relacionados y deberían estar juntos (cohesión).

Si cada vez que hacés una request pasas por 250 microservicios significa que está demasiado **acoplado (coupled)**.

Pregunta 3:

In an online Banking System architected as microservices, we have a microservice that handles the following three types of requests:

- 1) Requests for **transferring money** between accounts
- 2) Requests for **scheduling an in-person appointment** with a financial advisor
- 3) Requests for **opening** a new account or **closing** an existing account

What is the most correct statement about this microservice?

This microservice is too small since it handles only 3 types of requests.

This microservice is not cohesive enough.

This microservice breaks the **Single Responsibility Principle**.

Correcto

That is correct! Based on the requests it handles, the microservice has multiple unrelated responsibilities

This microservice doesn't correctly follow the Strangler Fig Pattern.

Microservices - Principles and Best Practices

Motivation for Database Per Microservice

Imaginemos por un segundo que tenemos un CustomersService y un ReportingService. Ambos usan la misma SQL Database pero el encargado de mantenerla es el CustomersService.

¿Qué sucede si CustomersService cambia algo a nivel de schema de la base? ¿Qué pasa si refactorizan 250 veces al día? Le tendría que avisar al Reporting Service las 250 veces para que ellos también refactoricen.

Esto es justamente lo que no queremos cuando tratamos con microservicios. Las cosas deberían ser totalmente independientes.

Benefits of Database Per Microservice

Cada microservicio tiene su propia base de datos y no la expone a ningún otro servicio.

El tradeoff es que hay latencia.

La idea es que si vos querés comunicar A con B entonces no usen la misma base sino que conectes a A con B a través de su propia API.

Ellos siempre te van a garantizar la respuesta de la misma manera y no importa que refactoricen el código 40000 veces, ellos siempre van a tener ese contrato con vos.

Downsides of Database per Microservice

1. Latencia: Al comunicar un microservicio con otro, hay latencia.
 - a. Mitigar latencia: Una de las estrategias comunes es que si A necesita de B, pidamos esa información y la cacheemos pero entra otro desafío. ¿Qué pasa si la información de A cacheada ya no es la real de B? Esto provocaría una pérdida de consistencia bastante grave.
 - i. Webhooks: A escuche cambios de B a través de un mensaje o webhook diciéndole que invalide.
 - ii. Tiempos de caché bajos (TTL): aunque la desventaja es que va a haber un tiempo que la información no sea consistente.
 - iii. Usar un caché distribuido: Que todos usen el mismo lugar para guardar cosas (Redis).
- La mejor idea suele ser usar webhooks o que A antes de devolver la respuesta, le pregunte a B si sigue siendo relevante la data.
2. No podemos usar JOIN's: Esto implica una degradación de performance. Para simular esto de la mejor forma se suele hacer query del microservicio A al B, y mapear la data para lo que necesitamos usando herramientas del lenguaje.
3. No tenemos la seguridad de las transacciones: En los monolitos a veces hacemos transacciones que modifican múltiples tablas, y que si alguna falla no hacemos el commit. Una **transacción distribuida** es **muy difícil** (y ni siquiera se usa)

DRY en el contexto de Microservicios

Definición

Es uno de los principios esenciales del Software Engineering.

Si estás usando la misma lógica o los mismos valores (constantes) en un mismo programa deberías considerar compartirlos en un único lugar a través de un método, una clase o una variable.

Si te está sucediendo lo anterior pero en múltiples aplicaciones, considerá armar una librería para que cualquier aplicación la pueda importar y usar.

Beneficios

Si hacemos un cambio lo hacemos una única vez.

Reducimos duplicidad.

El trabajo de un solo engineer puede ser reutilizado.

DRY no siempre es aplicable en Microservicios

El principio para utilizar microservicios es que todos estén **loose coupled** entre sí pero

1. ¿Qué sucede si usamos en todos los microservicios una misma librería y algo de esa librería cambia?
 - a. Cada uno de los microservicios que la utiliza debería cambiar también.
 - b. Al cambiar el uso en los microservicios, por ley, entonces debemos
 - i. Rebuildear
 - ii. Retestear
 - iii. Redeployear
2. ¿Qué sucede si hay un Bug/Vulnerabilidad en la librería? Impacta en **todos los microservicios** y rompe el principio de que los microservicios están aislados.
3. ¿Qué sucede si tenemos una librería A que usa una librería B con la versión 2.0 pero nosotros también usamos la librería B en el proyecto pero con la versión 1.0? No es posible.

- a. Esto se llama como **Dependency Hell**. Porque estamos indirectamente usando la misma librería con dos versiones diferentes. Para solucionarlo deberíamos garantizar que la librería B tenga la misma versión en todos lados y muchas veces no es la mejor idea (puede tener breaking changes)
- b. Otro problema es que estamos indirectamente **duplicando el código** y aunque tampoco parezca tan obvio, estamos haciendo más grande el build.

¿Cómo solucionamos los problemas de DRY?

Microservicios que comparten lógica

Si estamos usando la misma lógica de negocio y debemos compartirla entre microservicios significa que deberían ser un único microservicio.

No obstante, hay dos opciones

1. Mover todo lo que necesita esa lógica en el microservicio B al microservicio A.
2. Crear un nuevo microservicio C que se conecte con los microservicios A y B para entregarles lo que necesitan.

Microservicios que usan los mismos modelos de DB para comunicarse

Si ambos microservicios comparten el mismo lenguaje:

Tener librerías o archivos compartidos acá es una buena práctica. ¿Por qué? Porque uno requiere del otro.

Imaginemos que el microservicio A necesita usar User y Account, y el microservicio B también.

Si algo cambia de esos modelos, ambos deberían acoplarse porque el problema lo requiere. Si ambos tuviesen copias, y no tendrían la misma información, un test en un microservicio pasaría y en el otro no. Estallaría en producción.

¿Por qué esto es la excepción a nivel de acoplamiento? Porque el acoplamiento ya existe por naturaleza del negocio.

Si ambos microservicios NO comparten el mismo lenguaje: Es mejor usar un Code Generation desde un Data Schema.

Podemos usar especificaciones neutrales tipo JSON Schema, GraphQL Schema, Protobuf (gRPC) que son el contrato entre los microservicios.

Luego, cada microservicio **genera su propio código de modelos a partir del schema**, usando herramientas de **codegen**.

Código que está bien duplicarlo

Para **utility methods** que cambian constantemente, es mejor tener la propia implementación dentro de cada microservicio.

Cuando hablamos de **utility methods** son métodos que NO forman parte directa de la lógica de negocio, pero que son necesarias para que el código funcione. Por ejemplo, los utils o helpers.

Las razones son las siguientes

1. Cada microservicio tiene su implementación optimizada propia.
2. Permite migrar los microservicios a otros lenguajes de programación

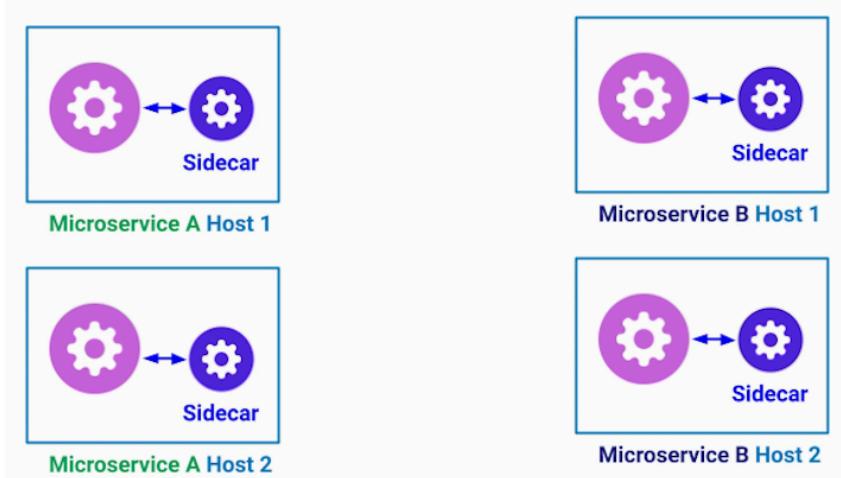
Como duplicar código (sin duplicarlo tan gravemente)

Sidecar Pattern

Es un patrón de arquitectura en el cual si tenemos el microservicio A, y un componente auxiliar, los podemos conectar sin modificar el código principal.

Se llama sidecar (como la moto con un asiento al costado) porque:

1. No es parte del motor (la lógica del negocio del servicio).
2. Va **al lado** del servicio principal, corriendo en el mismo host o pod y se comunica con él a través de HTTP.



Si bien tenemos una pequeña latencia no es tan grave como si se conectara con otro host **pero** la latencia es mayor que si tuviésemos una librería.

Usar una librería

Cuando tenemos código genérico que es estable y no suele cambiar como podría ser para Logging, Retrying, o Pattern Matching podemos agregarlo en el código del microservicio

Importante: Esto tiene que ser usado como último recurso.

Notas finales para DRY en microservicios

1. En microservicios siempre tenemos que seguir DRY.
2. La duplicidad de código es **inaceptable**.

Duplicación de información en Microservicios

Imaginemos que tenemos dos bases de datos diferentes, una para cada microservicio A y B pero tienen la misma información.

Para empezar, no debería suceder que ambos tengan la misma información. Sino, que el microservicio A debería exponer al microservicio B una API para obtener esos datos y si se quiere mejorar la velocidad, usar caché.

Ahora sí, si queremos prescindir de hacer la llamada de API por razones de rendimiento y queremos “tener la misma base de datos” tenemos algunas consideraciones

1. Debe haber un dueño de la información definido. Solo uno debe ser la fuente de la verdad.

- a. Esto quiere decir que si el Microservicio A tiene la misma información que la base en el Microservicio B debe suceder que:
 - i. El Microservicio A es quien Escribe/Actualiza/Elimina → Replica la información a la base que usa B.
 - ii. El Microservicio B solo puede leer de su propia base de datos (que escucha los datos replicados desde la base del microservicio A).
2. Solo podemos garantizar una consistencia eventual. Puede ser que en algún momento, estando duplicando la información, alguna base esté desfasada de la otra por un tiempo. En algunos casos es aceptable y en otros casos no.
 - a. Caso aceptable: Digamos que tenemos un ecommerce y tenemos un ProductsService y un ReviewsService. A la hora de que el cliente quiere ver un Producto, lo común es pedir las Reviews al ReviewsService pero esto puede tomar un tiempo (mensajería o conexión a través de HTTP). Como esta información no es crucial ni tampoco crítica podemos cachearla en el ProductsService para evitar pedirle la info a ReviewsService, no obstante, debido a la naturaleza del caché, puede desfasarse en algún momento.
 - b. Caso no aceptable: Si tenemos que actualizar el balance de un usuario (UserService) o el stock de un producto (InventoryService) determinado NO podemos aceptar duplicar cosas, sino que la consistencia es estricta. Lo que nos dice que NINGUN MICROSERVICIO deberá CACHEAR ni DUPLICAR esta información.

Determinar autonomía estructurada para equipos

Problema con un equipo que tiene máxima autonomía

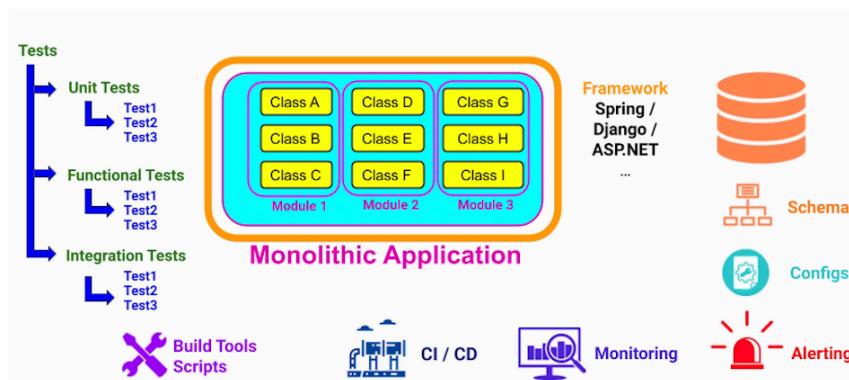
Existe un mito que dice que el mayor beneficio de los microservicios es que cada equipo puede elegir su propio

1. Stack

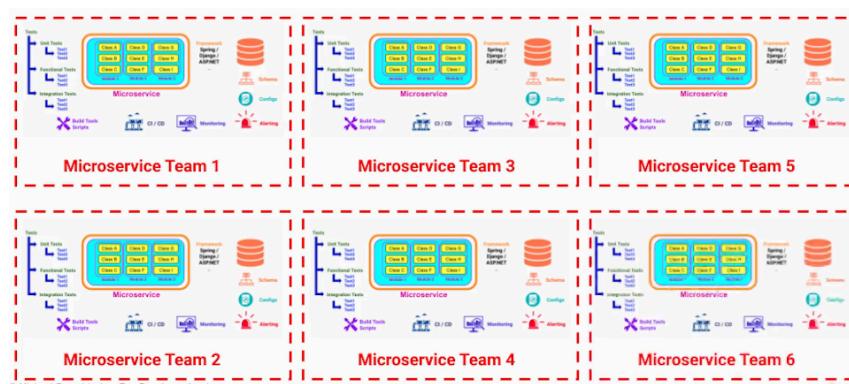
2. Herramientas
3. Bases de datos
4. API
5. Frameworks

Esto es **erróneo**.

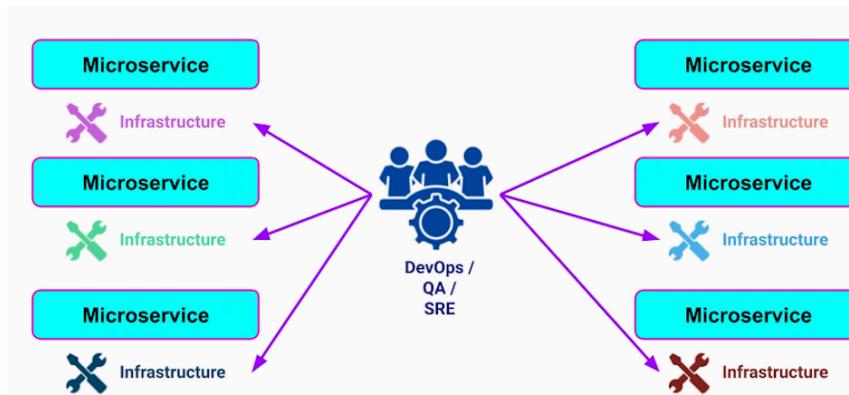
1. Porque cuando vamos a migrar a microservicios significa que hicimos un gran producto monolítico el cual tuvimos que pensar y definir las siguientes cosas.



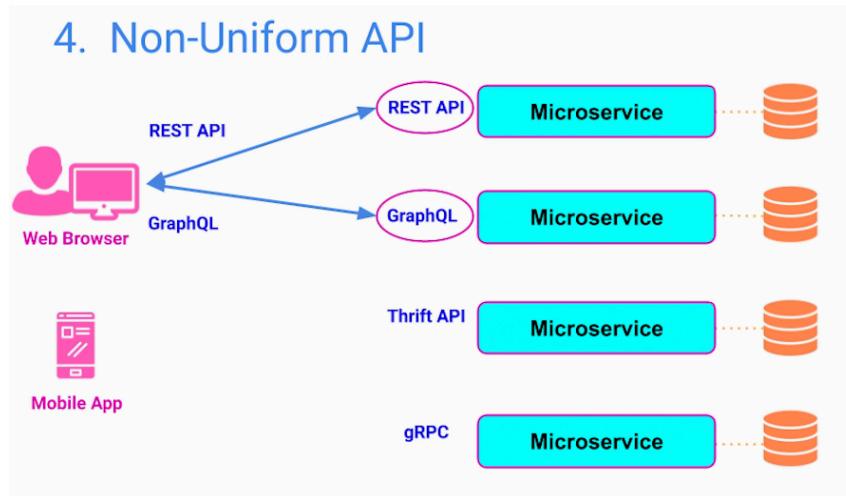
Si cada equipo tendría que hacer esto mismo en cada uno de sus microservicios (acoplando todas las herramientas, los schemas, las configuraciones, el monitoreo, los scripts de build, etc) estaríamos gastando mucho tiempo e infraestructura.



2. Mantener toda esta infraestructura, en cada lenguaje y cada herramienta particular es un dolor de cabeza.



3. Si un desarrollador va a desarrollar en más de un microservicio, la curva de aprendizaje es más empinada. Porque tiene que aprender cada lenguaje y herramientas que usa ese microservicio.
4. Se convierte en una Non-Uniform API. Recordemos que la idea de hacer microservicios es separar una estructura monolítica.
 - a. Si cada equipo define la API a su manera, y el frontend manda las peticiones al API Gateway para que lo redirija al microservicio correspondiente, sería un dolor de cabeza para él que todos los microservicios sigan su propio estándar de respuesta o convenciones.
 - b. Lo mismo que a. si tenemos que conectar microservicios entre sí.



Pero ¿no era que la idea es que los microservicios puedan usar sus propios lenguajes y que eran increíbles por eso? Sí y no. En este tipo de arquitectura hay un balance entre **autonomía y estructura** (Structured Autonomy).

Límites de autonomía para cada equipo

Tier 1 - la más restrictiva

Deben ser un estándar para toda la empresa.

1. Infraestructura: Esto nos permite invertir mucho esfuerzo por adelantado en adoptar o crearla desde 0.
 - a. Monitoreo y alertas
 - b. CI/CD
2. API guidelines & best practices para definir API públicas y privadas.
 - a. A los clientes no les importa qué servicio es el que les responde. Ellos esperan un estándar.
 - b. Es más fácil de trabajar para los equipos de los microservicios comunicándose.
3. Seguridad & políticas de cumplimiento de datos: Si alguno de los microservicios es hackeado, y ese microservicio no era seguro, exponemos mucha información y nuestra organización será responsable.

Tier 2 - Libertad con límites

1. Elegir lenguajes de programación
2. Bases de datos a utilizar

Al fin y al cabo, cada solución de tiempo de ejecución y almacenamiento de datos es óptima para un caso de uso diferente.

No obstante, las tecnologías (lenguajes/bases de datos) deberían estar limitadas por la organización. Si no se permite usar C, no podés usar C aunque quieras. Esto es porque se quiere evitar una jungla de tecnologías y que todo el mundo tenga que aprender 250 tecnologías diferentes.

Tier 3 - Completa autonomía

1. Lanzan features en función de sus prioridades que consideran relevantes.
2. Pueden organizar su propio calendario y la frecuencia.
3. Pueden desarrollar scripts para desarrollo local o testing con el fin de que les aporte productividad a ellos.
4. Son dueños de su propia documentación.
5. Proceso de onboarding de nuevos desarrolladores.

Factores importantes para los límites en los Tiers

1. Si el equipo / influencia de DevOps es grande y ellos piden que no se usen más de X cosa por Y cosa, se les debe respetar.
2. Seniority de los desarrolladores: Cuanto más seniors son los desarrolladores, más libertad prefieren a la hora de crear o construir su propia infraestructura.
3. Cultura de la empresa: Algunas empresas solo te dejan usar 1 único lenguaje de programación. La ventaja de esto es que podés contratar a los desarrolladores una vez y moverlos de un lado a otro.

Arquitectura de Microservicios en frontend (micro-frontends)

Problema de frontend un monolítico

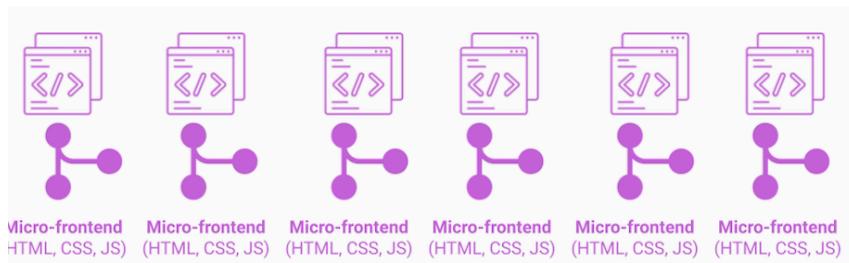
1. A medida que el código crece se hace más difícil de mantener.
2. Cuesta más hacer testing y deployear.
3. A la hora que el frontend quiere implementar algo, tiene que averiguar quienes son los responsables en el backend de ese microservicio y decirles para que lo implementen.
4. Si el backend implementó algo y el frontend tiene que cambiar algo, hay que hacer un redeploy entero de toda la aplicación.

Los puntos 3 y 4 muestran un acoplamiento muy fuerte entre ambos equipos.

Aplicación monolítica a microfrontend

Dividimos la aplicación web monolítica en múltiples módulos frontales o bibliotecas que actúan como SPA independientes.

La división se puede hacer por capacidad de negocio o dominio al igual que hicimos en los microservicios.



Cada página puede tener más de un micro-frontend visible en la misma página, los cuales están totalmente desacoplados uno del otro. Estos se pueden cargar como una aplicación de web para fines de testing.

Para poder ser ensamblados, se hace en tiempo de ejecución mediante una aplicación contenedora que se ejecuta cuando el usuario entra a través del navegador.

Roles de la Aplicación Contenedora

1. Renderizar elementos comunes.
2. Hacerse cargo de las funcionalidades comunes y librerías.
3. Decirle a cada frontend cuando y donde deben ser renderizados.

Confusiones comunes

1. Micro-frontend requiere a un patrón de arquitectura, no a un framework.
2. Los micro-frontend no son elementos de UI reusables. Son una aplicación específica que resuelve un problema del negocio.

Funcionamiento de una aplicación micro-frontend + backend microservicios

1. El usuario solicita la página.
2. La aplicación web le devuelve la Aplicación Contenedora.
3. La aplicación contenedora gestiona la autenticación con el Users Service (backend).
4. La aplicación contenedora almacena el token de autenticación del usuario en su dispositivo.
5. La aplicación contenedora renderiza el header y footer de la página.
6. La aplicación contenedora llama a los micro-frontends.

7. Cada micro-frontend solicita a cada microservicio que corresponde el contenido.
8. Cada microservicio le devuelve la información a cada micro-frontend.
9. El micro-frontend genera el HTML y lo pone donde le haya dicho la aplicación contenedora.
10. El usuario solicita otra página, la aplicación contenedora desmontará los micro-frontends que están ocupando memoria en esa página.
11. Repite paso 1.

Beneficios de Micro-Frontends

1. Separás un monolito complejo a algo más pequeño y manejable.
2. Cada equipo mantiene su propio micro-frontend.
3. Es más fácil y rápido de testear de manera aislada.
4. Cada uno tiene su pipeline CI/CD.
5. Cada equipo maneja su propio calendario.

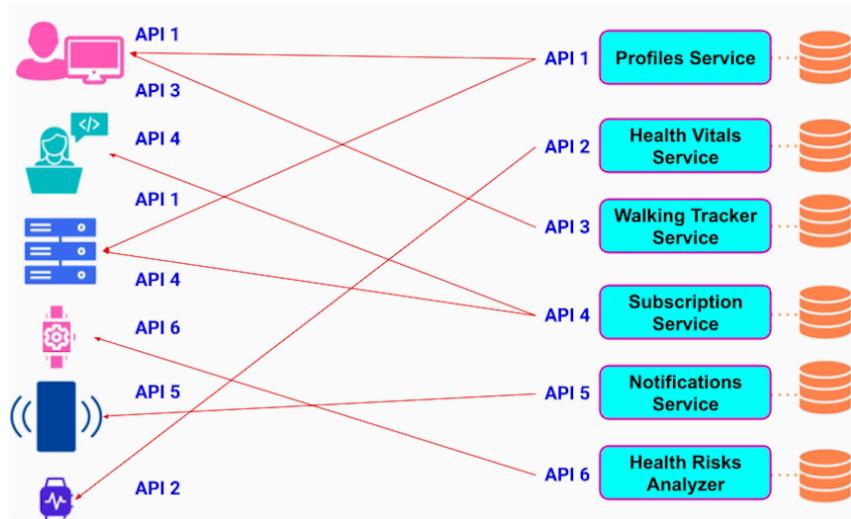
Mejores prácticas para Micro-Frontends

1. Cargarlos en tiempos de ejecución. No deben ser expresados como dependencias de la Aplicación Contenedora.
2. No comparten estados en el navegador.
3. Para comunicarse entre sí pueden usar:
 - a. Eventos personalizados
 - b. Callbacks
 - c. Comunicarse por la barra de URL.

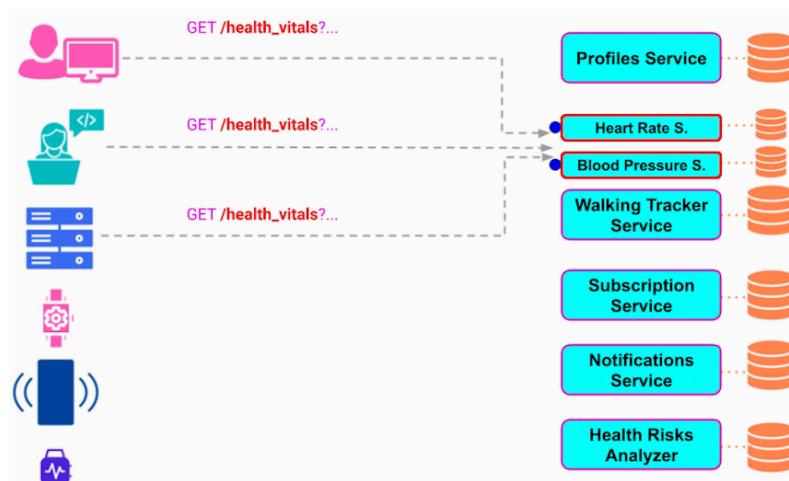
Manejo de API para Arquitectura de Microservicios

El problema de manejar APIs

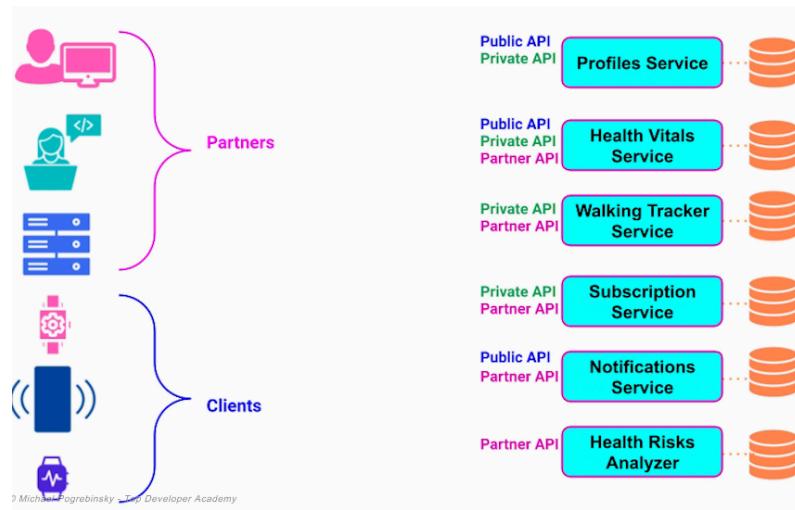
1. Usar directamente conexión de cliente a microservicio los acopla muchísimo.



¿Qué pasaría si ahora el microservicio (API 2) se separa en dos? Los clientes deberían notar esto y refactorizar todos.

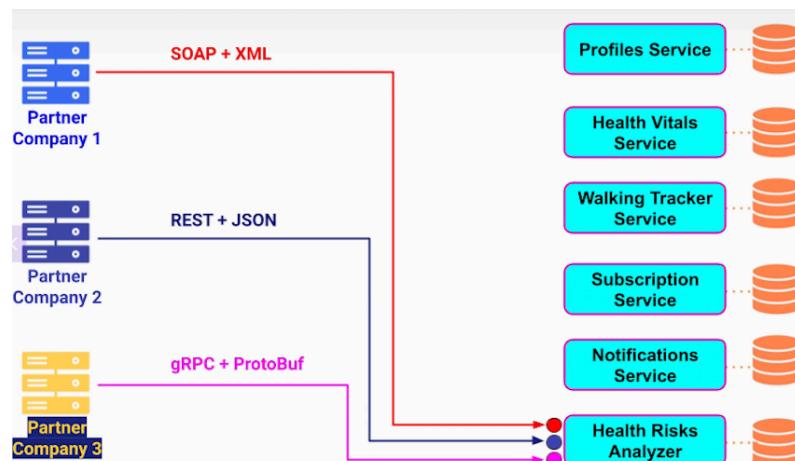


2. Diferentes tipos de API para diferentes consumidores (public/private/partner)

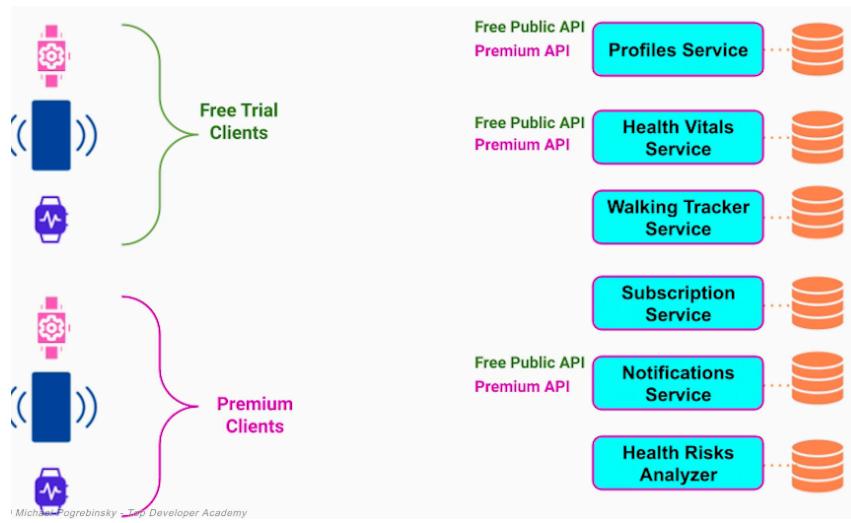


Exponemos los Public API endpoints a los clientes, Private API endpoints para que los microservicios se hablen internamente, y Partner API endpoints solamente para nuestros partners.

Internamente, podemos mantener la consistencia para los private api endpoints a nivel de input/output. Pero no es tan simple hacerlo para Private/Partner APIs. ¿Por qué? Es posible que la compañía que nos va a mandar la petición tiene su propio sistema y nosotros debemos recibirla con ese formato.



3. Diferentes tiers de API basado en subscripciones.



4. Control de tráfico y monitoreo: Cada microservicio debería tener control del rate limiting + autenticación. Supongamos que un cliente mobile para renderizar la home necesita llamar a 3 microservicios. Para cada microservicio debería autenticarse y ver que no se haya excedido el rate limiting. Esto es costoso y complicado de trackear.
5. Esfuerzo duplicado en los microservicios.

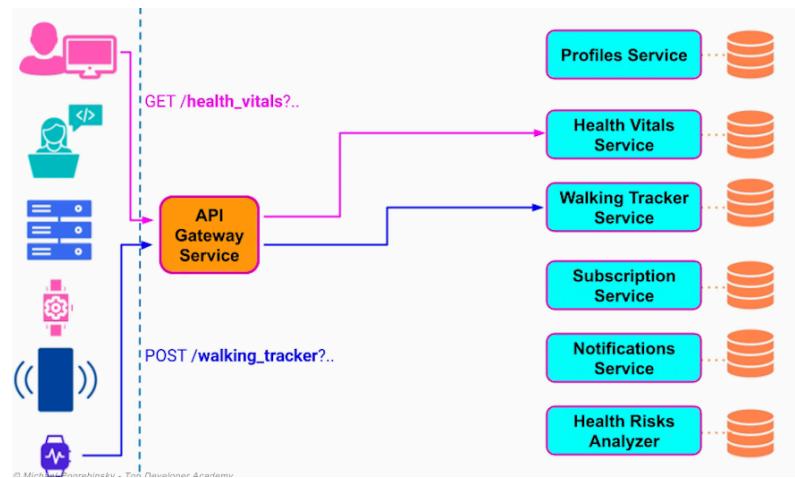
API Gateway Pattern

Nos permite desacoplar el client-side de la arquitectura interna produciendo que el API Management sea más fácil.

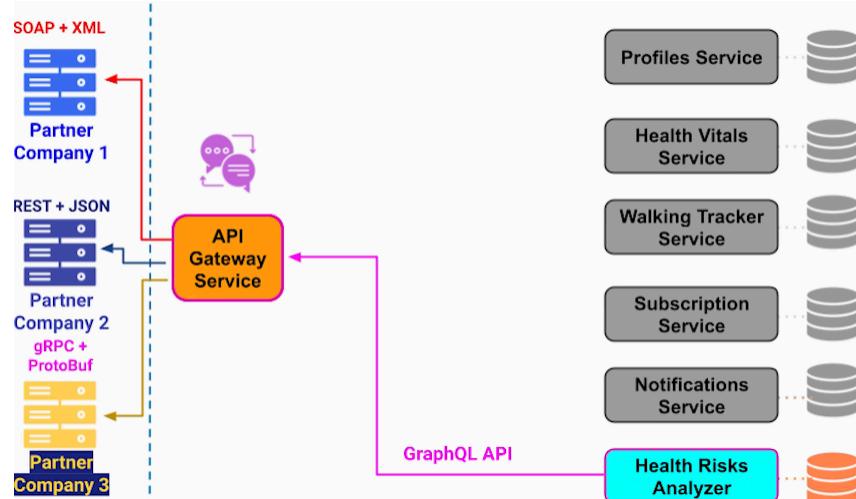
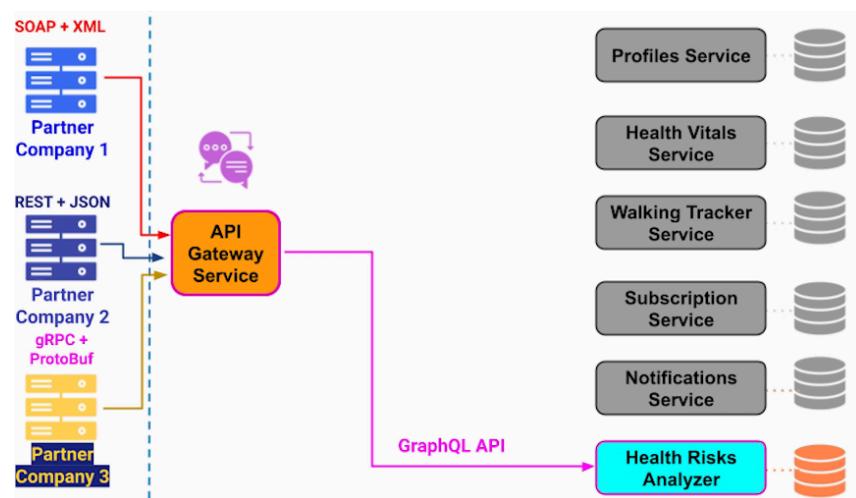
Es prácticamente obligatorio usarlo en la arquitectura de microservicios aunque se puede usar en otras arquitecturas.

El API gateway:

1. Recibe las solicitudes y este las redirige a cada microservicio correspondiente.



- Transforma información: Si clientes nos mandan información en un formato diferente (*porque usan difs. tecnologías*) nosotros podemos parsearla y mandarla en el formato que nos pide el microservicio (o viceversa, el *API_GATEWAY parsear la data al formato deseado por el cliente*).



3. Puede hacer manejo de tráfico o **throttling**: La idea es que el cliente no pase del API GATEWAY si excedió su tier. El objetivo es que no saturen a los microservicios para clientes que quieren usarlo de forma normal.
4. Puede manejar temas de autenticación (*chequear que el token sea válido, refrescar el token*). Es importante notar que el microservicio aún deberá validar que el token sea válido. De lo contrario si alguien encuentra el microservicio y no pasó por el API Gateway estaría expuesta.
5. Permite supervisar todo el tráfico a nuestros microservicios desde un único punto de entrada. Esto nos permite detectar problemas en APIs concretas o analizar el comportamiento de nuestros clientes y empresas asociadas.
6. Permite distribuir múltiples requests de un cliente a multiples microservicios, y cuando todas están resueltas, el API Gateway devuelve el resultado.

Load Balancer vs API Gateway

Similitudes

Redirigen una request a un único destino.

Diferencias

1. Load Balancer: Equilibra la carga de tráfico en un grupo de servidores. En el contexto de microservicios, se suelen poner para redistribuir el tráfico en varias instancias de un mismo microservicio.
2. API Gateway: Detecta a qué servicio querés ir, y te redirige. Antes, de llegar al servicio que querés te encontrás con el Load Balancer (1) de ese microservicio, y según cual sea la carga de las instancias, te manda a una instancia u otra.

Características de un Load Balancer

1. Evita que haya sobrecarga de servicios.
2. Sirve para hacer chequeos frecuentes de salud de ciertos servicios.
3. Ofrece diferentes routing algorithms para entender cuál es la mejor manera para llegar a un lugar particular (gracias a la URL, headers, carga, u otras reglas).

Características de un API Gateway

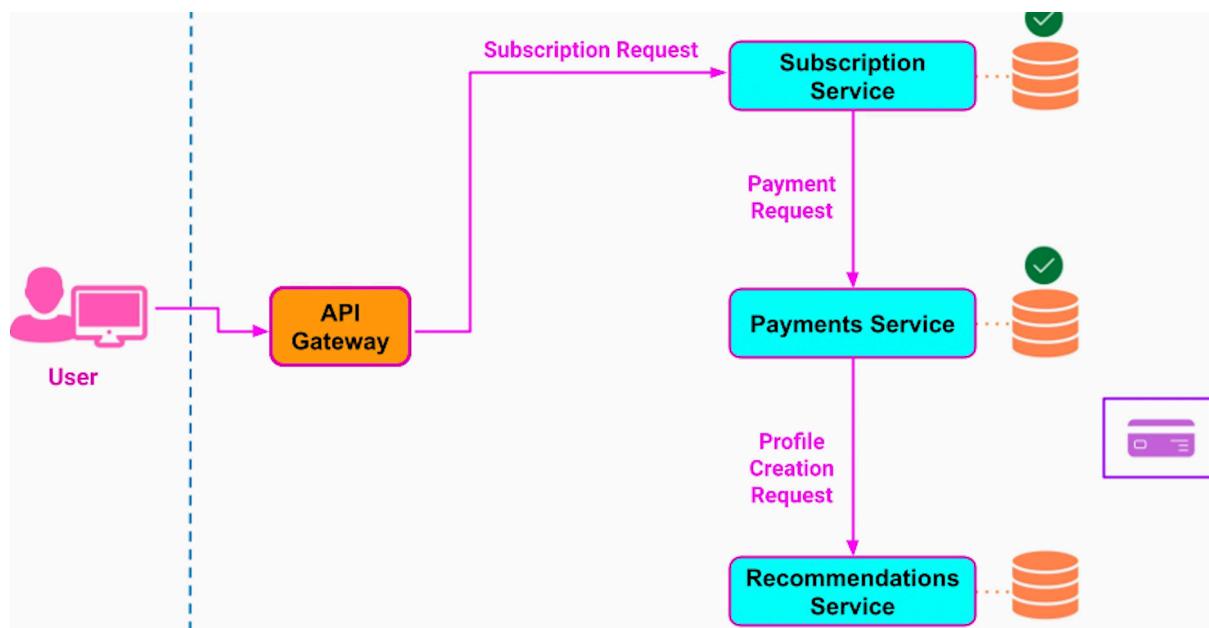
1. Throttling.
2. Monitoreo.
3. Versionado de API & manejo.
4. Protocolos y transformación de información.

Introducción a Event-Driven Architecture

Motivación

Imaginemos que una persona se quiere subscribir a nuestra aplicación. Si tenemos una arquitectura orientada a microservicios va a suceder que tiene que pasar por varios servicios para que efectivamente la subscripción se active.

Intento 1

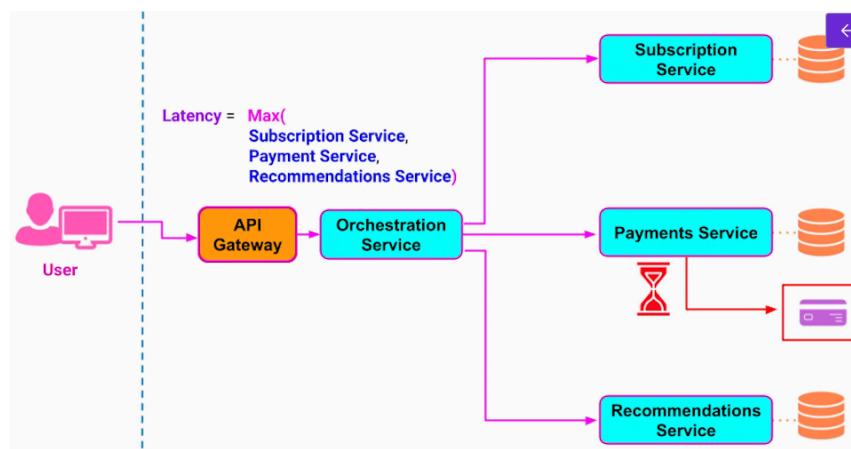


¿Cuál es el problema de esto?

1. La latencia de conectar cada servicio con el otro. La latencia total es la suma de la latencia que toma cada servicio.
2. El manejo de errores:

- ¿Qué pasa si el Payments Service se muere porque se cayó el servicio de pagos?
- ¿Qué pasa si el PaymentsService funcionó pero el Recommendations Service no, que le decimos al usuario de sus recomendaciones?
- ¿Qué pasa si prescindimos del RecommendationsService porque quizá tarda mucho y solo dejamos el SubscriptionService y PaymentsService? Estaríamos teniendo inconsistencias de información.

Intento 2



¿Cuál es el problema de esto?

1. Mejoramos la latencia porque ahora el OrchestrationService redirige los procesos para realizarse en paralelo, lo cual el tiempo total del proceso será el máximo de los tiempos de cada servicio.
2. ¿Qué pasa si algún servicio se parte en dos? El OrchestrationService está completamente acoplado, lo cual deberíamos refactorizar y testear.

No resolvimos nada.

Event-Driven Architecture

Evento

Es el concepto fundamental de este tipo de arquitectura.

1. Representa un hecho, una acción o cambio de estado.
2. Es inmutable.
3. Puede ser almacenado indefinidamente (las requests no).

4. Pueden ser consumidos varias veces por diferentes servicios (las requests solo pueden ser consumidas una vez por un único servidor).

Participantes

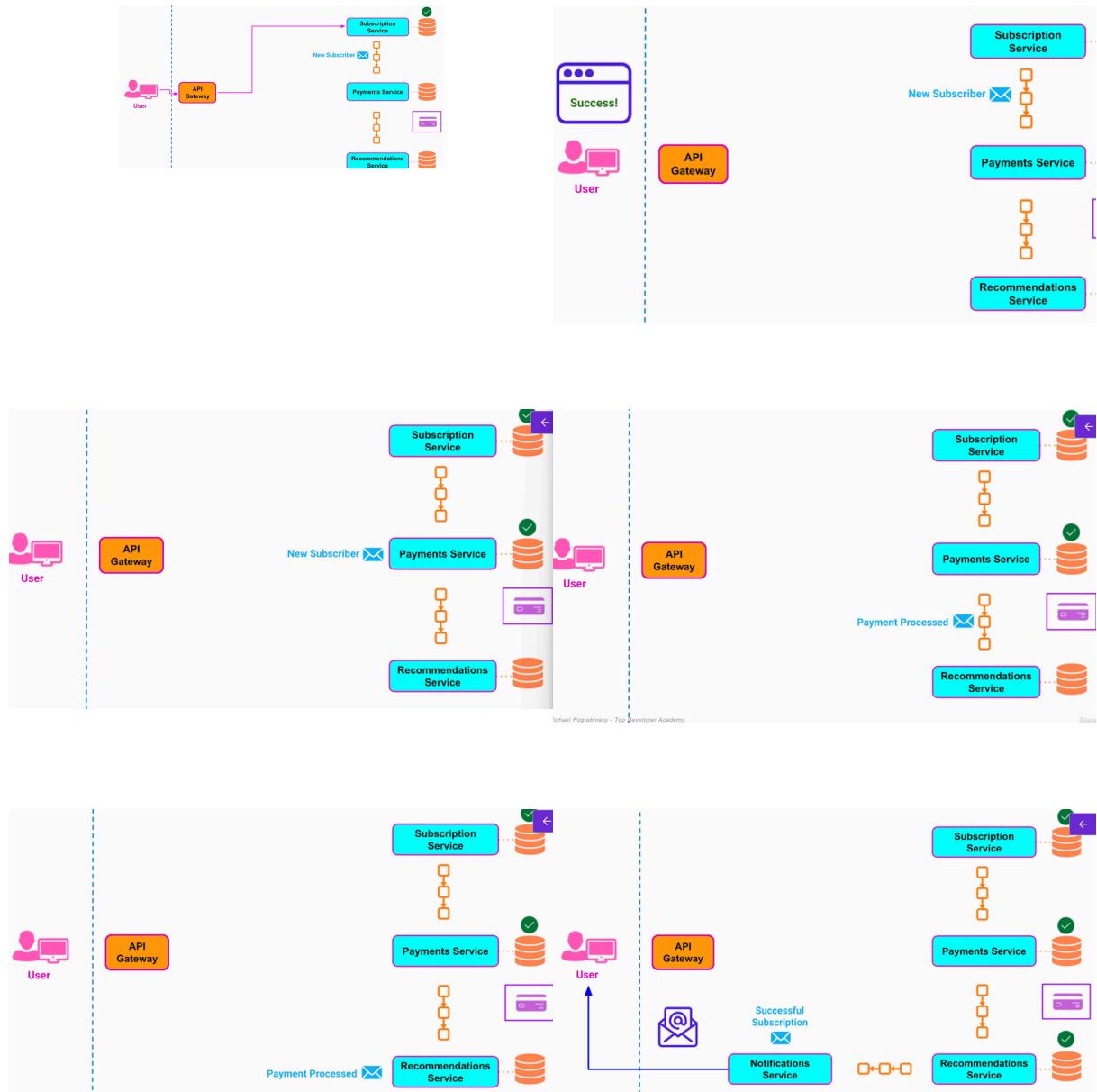
1. Producer: Es quien emite el evento.
2. Message-Broker: Almacena y rutea esos eventos.
 - a. Puede emitir un evento a la cantidad de consumers que quiera.
 - b. El evento que llega queda almacenado de forma confiable. Incluso si el consumidor que debía procesarlo falla, se puede volver a leer desde el broker más tarde.
 - i. A esto le llamamos redundancia planificada.
3. Consumers: Reciben y procesan los eventos.

Request-Response vs Event-Driven

1. La comunicación de un modelo Request-Response es síncrono mientras que el dirigido por eventos es asíncrono.
 - a. En el modelo de Request-Response, si el Sender manda una petición tiene que esperar que el Receiver responda aunque tenga información útil o no. Si la respuesta nunca llega, se queda colgado.
 - b. En el modelo de Event-Driven, el Producer no espera ninguna respuesta del Consumer ni tampoco de los Consumers del evento. Esto permite al Producer pasar a procesar su siguiente tarea en lugar de esperar una respuesta que capaz quizá no necesita.
2. Inversion of Control
 - a. En el modelo de Request-Response, el Sender necesita saber exactamente quién va a recibir la petición y cómo. Si quiere contactarse con más de uno al mismo tiempo, debe esperar cada respuesta y conocer sus parámetros únicos. Esto produce que el Sender esté acoplado a los servicios.
 - b. En el modelo de Event-Driven Model, al Producer no le interesa y puede que ni siquiera conozca a aquellos servicios que van a consumir el evento.

3. Loose Coupling: Esta es la razón por la cual la Event-Driven Architecture va tan de la mano con la arquitectura de microservicios.

Corrigiendo el problema que planteamos en la motivación a Event-Driven



Ojo: Acá queda medio ambiguo. Si PaymentService fuese el servicio que procesa si el pago es real, no sé si devolvería en SubscriptionService el "success" al usuario porque ni siquiera sabés si la subscripción va a poder activarse o no. En tal caso, la conexión entre SubscriptionService y

PaymentService la haría mediante HTTP de forma async para garantizar que primero me confirmen si todo salió bien o mal.

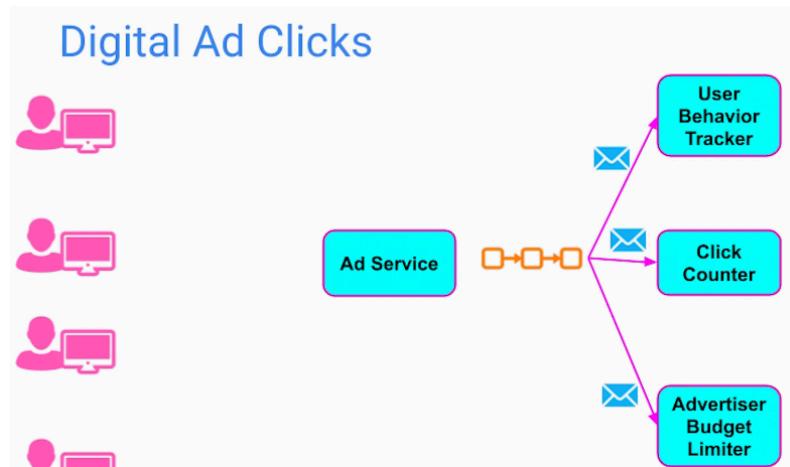
Si salió todo bien ahí sí inmediatamente le devuelvo “success” al usuario y el resto si lo hago asíncronamente con los message brokers. De lo contrario, el mensaje de “success” no es tan real.

Casos de uso y patrones para una arquitectura Event-Driven

Event-Driven Architecture use cases

1. Fire and Forget: En casos donde el cliente no espera una respuesta o no espera una respuesta inmediata.
 - a. Esperás respuesta (no inmediata): Cuando querés generar un reporte pesado que no sabés cuánto va a demorar. La idea es que lo generes y le mandes una notificación al usuario cuando el proceso haya terminado.
 - b. No esperás respuesta: Cuando un usuario deja una review en un producto por ejemplo, al usuario no le importa que le notifiquemos cuando haya dejado la review, sino solamente dejarla.
2. Entrega confiable: Cuando no nos podemos permitir no recibir una acción.
 - a. En transacciones financieras no podemos perder ningún mensaje.
 - b. Si una persona pagó un producto pero no notificamos al servicio de entregas porque se cayó, no podemos permitirnos no saber qué sucedió. De alguna manera tenemos que seguir enviandolo infinitamente hasta que se reciba.
3. Entrega infinita de eventos: En información de ubicación de tiempo real o de sensores, mandamos tanta información cada milisegundo que no podemos permitir bloquearnos. Ni siquiera nuestro sistema debería ponerse a pensar qué hacer, ni tampoco procesarlos. Solamente deberíamos recibirlas y encollarlas para un posterior análisis.
4. Detección de anomalías o reconocimiento de patrones: Si el broker no está recibiendo mensajes algún problema con los Producers está habiendo.

5. Broadcasting: Cuando queremos enviar la noticia de que un cliente hizo X a otros servicios, pero ese cliente ni siquiera se enteró de que efectos secundarios acaba de producir. Esto se suele usar en publicidades de sitios web para cobrar.



Cuando recibiste un click, tu AdService a través de un MessageBroker le comunica a los demás servicios lo que sucedió pero el cliente ni se enteró.

6. Buffering: Podemos tolerar una masiva cantidad de eventos viniendo de un único servicio poniendo un message-broker y comunicar asíncronamente los servicios.

- a. Si tenés una red social, y un post se hace super viral a tal manera que responden 1 millón de personas a la vez, en vez de hacer la recepción y mandar directo a PostsService y CommentsService de un saque ponés en el medio un Message Broker para recibir todos esos mensajes y poder ir derivando a los servicios a medida que pueden procesarlos. Esto te permite recibir todos los mensajes pero no sobrecargar los servicios

Request-Response model use-cases

1. Cuando el usuario quiere la respuesta YA
 - a. Si tenemos una tienda donde vendemos productos, antes de ingresar a la página los productos tienen que estar cargados. De lo contrario, abandonarán el sitio.
2. Cuando la interacción es tan simple que no necesitamos un modelo Event-Driven. Esto viene por el lado de que manejar un Message-Broker,

hostearlo y mantenerlo tiene un costo.

En la realidad

Una arquitectura de Microservicios combina:

1. Event-Driven Architecture
2. Synchronous request-response model

Es mejor:

1. Empezar con un request-response model.
2. Upgradear a event-driven architecture cuando **solo sea necesario**.

Event-Delivery Patterns

Event-Streaming

En este patrón, el message-broker es utilizado como almacenamiento temporal o permanente para eventos.

Los consumers tienen full-access a los logs de esos eventos, incluso si ya fueron consumidos por el mismo consumer o por otros.

Este patrón es una gran elección para los siguientes casos de uso:

1. Reliable delivery (entrega segura): debido a que el message-broker o bien tiene los eventos indefinidamente o bien los tiene durante un largo periodo de tiempo tal que nos permite acceder a esos eventos y auditarlos si fuese necesario.
2. Pattern / Anomaly detection: Pues el consumer necesita acceso a todos los eventos pasados en una ventana concreta.

Pub/Sub

En este patrón, los Consumers se suscriben a una queue particular o canal para recibir nuevos eventos luego de subscribirse.

En este caso, los subscriptores **no tienen acceso a eventos viejos**, y tan pronto como los subscriptores actuales reciben el evento, el message-broker lo borrará de su queue.

Este patrón es una gran elección para los siguientes casos de uso:

1. El message broker está siendo utilizado como un almacenamiento temporal o como mecanismo de broadcasting: Luego de que los suscriptores consumen los eventos, estos tipicamente son transformados y almacenados permanentemente en una base de datos o son pasados a otro servicio.
2. Fire and Forget
3. Broadcasting
4. Buffering
5. Infinite stream of events

Semántica de Entrega de Mensajes en Event-Driven

Releer, es re interesante

In an IOT system that manages autonomous home vacuum cleaners, we have **3 types of events:**

Event Type A - Location update from a vacuum cleaner to our system (this event is sent automatically every 2 seconds)

Event Type B - Malfunction of a vacuum cleaner that needs the attention of the owner

Event Type C - Payment from the owner of the vacuum cleaner for our cloud-based services.

What is the best **delivery semantics** for each type of event?

Event Type A - At Most Once

Event Type B - At Least Once

Correcto ✓

Event Type C - Exactly Once

That is correct! It's OK to lose location updates, but we cannot afford to lose an event notifying us about a malfunction. We also have to process the payment once and only once.

Pregunta 3:

What is the difference between Event Streaming and Pub/Sub Patterns for Event-Driven Architecture?

In Event Streaming, consumers have access to old events, while in Pub/Sub, consumers only get notified about new events as they arrive.

Correcto ✓